Exercícios

Exercício 1: Email Notification

Refatore o código para que Notificacao não dependa diretamente de EmailService, mas sim de uma abstração (interface).

- **Inversão de Dependência** *(DIP)*

```
class EmailService {
    public void enviarEmail(String mensagem) {
        System.out.println("Enviando e-mail: " + mensagem);
    }
}

class Notificacao {
    private EmailService emailService = new EmailService();
    public void enviarNotificacao(String mensagem) {
        emailService.enviarEmail(mensagem);
    }
}
```

Exercício 2: Operador de Máquina
Refatore o código aplicando e separe as responsabilidades para que um trabalhador não seja forçado a implementar métodos que não fazem sentido para ele.

- **Segregação de Interface** *(ISP)*

```
class OperadorDeMaquina implements Trabalhador {
    @Override
    public void trabalhar() {
        System.out.println("Operador está trabalhando...");
    }

    @Override
    public void fazerPausa() {
        System.out.println("Operador está fazendo uma pausa...");
    }

    @Override
    public void programar() {
        throw new UnsupportedOperationException("Operadores de
máquina não programam!");
    }
}
```

Exercícios

SOLID - ISP e DIP

Exercício 3: Funcionário

A interface Funcionario viola o *ISP* porque força implementações desnecessárias (como *dirigir* para o **Desenvolvedor** e *programar* para o **Motorista**).

- Refatore o código para que a interface seja mais coesa, criando interfaces menores e mais específicas para as responsabilidades de cada tipo de funcionário.
- Aplique o *DIP* para que as classes **Desenvolvedor** e **Motorista** dependam de abstrações mais específicas, não de uma interface única que combine todas as responsabilidades.

```
interface Trabalhador {
    void trabalhar();
    void fazerPausa();
    void programar();
}

interface Funcionario {
    void baterPonto();
    void calcularSalario();
    void programar();
    void dirigir();
}

class Desenvolvedor implements Funcionario {
    public void baterPonto() {
        System.out.println("Desenvolvedor bateu ponto.");
    }

    public void calcularSalario() {
        System.out.println("Calculando salário do desenvolvedor.");
    }

    public void programar() {
        System.out.println("Programando...");
    }

    public void dirigir() {
        throw new UnsupportedOperationException("Desenvolvedores não dirigem no trabalho!");
    }
}

class Motorista implements Funcionario {
    public void baterPonto() {
        System.out.println("Motorista bateu ponto.");
    }

    public void calcularSalario() {
        System.out.println("Calculando salário do motorista.");
    }

    public void programar() {
        throw new UnsupportedOperationException("Motoristas não programam!");
    }

    public void dirigir() {
        System.out.println("Dirigindo...");
    }
}
```

Exercício 4: Gestão de Produtos MarketPlace
Na Amazon, os clientes podem escolher entre diferentes opções de entrega para seus pedidos:

- **Entrega padrão** (frete normal).
- **Entrega expressa** (mais rápida, mas mais cara).
- **Entrega programada** (o cliente escolhe um horário específico).
- **Retirada em locker da Amazon** (o cliente busca o produto em um ponto de coleta).

O sistema atual tem uma única classe *CalculadoraFrete*, que possui toda a lógica de cálculo de frete dentro dela, dificultando a manutenção e a adição de novos métodos de entrega.

Tarefa:
Codifique para permitir a inclusão de novos tipos de entrega sem alterar a lógica central do sistema.

Exercício 5: Gestão de Produtos MarketPlace
Os vendedores do site OlX podem listar diferentes tipos de produtos, como:

- **Produtos físicos** (celulares, roupas, livros).
- **Produtos digitais** (e-books, cursos online).
- **Serviços** (instalação de TV, consultoria).

O sistema atual usa uma interface **Produto** com métodos como *calcularFrete()* e *controlarEstoque()*, mas nem todos os produtos precisam dessas funcionalidades (por exemplo, serviços não têm frete ou estoque).

Tarefa:
Codifique para que cada tipo de produto tenha apenas os métodos necessários, evitando que classes sejam obrigadas a implementar funcionalidades que não usam.

Exercícios

SOLID - ISP e DIP

Desafio - Exercício 6: Gestão de Produtos MarketPlace

Durante a Black Friday, a Amazon precisa de um sistema para gerenciar produtos e aplicar diferentes tipos de descontos, sem impactar a estrutura principal do código.

Os descontos podem variar, como:

- **Desconto percentual** (ex: 20% de desconto em eletrônicos)
- **Desconto fixo** (ex: R$50 de desconto em compras acima de R$500)
- **Frete grátis** para determinadas categorias de produtos

Além disso, cada produto pode ter atributos diferentes. Por exemplo:

- Eletrônicos têm garantia e voltagem.
- Livros possuem autor e ISBN.
- Roupas têm tamanho e cor.

Dica: Para resolver os desafios, você precisará aplicar **todos os princípios SOLID**:

S - Separe bem as responsabilidades das classes (evite que uma única classe faça tudo).

O - Permita que novas funcionalidades sejam adicionadas sem modificar o código existente.

L - Certifique-se de que subclasses possam substituir as classes-pai sem problemas.

I - Divida interfaces para que classes implementem apenas o necessário.

D - Use abstrações para evitar dependência direta de classes concretas.

Desafio:

1. O sistema deve permitir adicionar novos tipos de desconto sem modificar as classes principais.
2. Deve ser possível adicionar novos tipos de produtos sem precisar alterar o código de produtos já existentes.
3. O código deve ser flexível e escalável para futuras promoções.